Agreement

Commit protocols

Atomic commitment protocols

Atomic commitment protocols are a form of agreement widely used in database management systems.

Atomic commitment **ensures atomicity**: a transaction can only commit or abort: if it commits, its updates are durable; otherwise, there are no side effects. Furthermore, consistency relies on atomicity as well.

If the transaction updates data on multiple node, either all nodes commit, or all nodes abort that transaction. This also means that if any node crashes, all must abort.

Consensus vs. atomic commitment

Here are the main differences between consensus protocols and atomic commitment protocols:

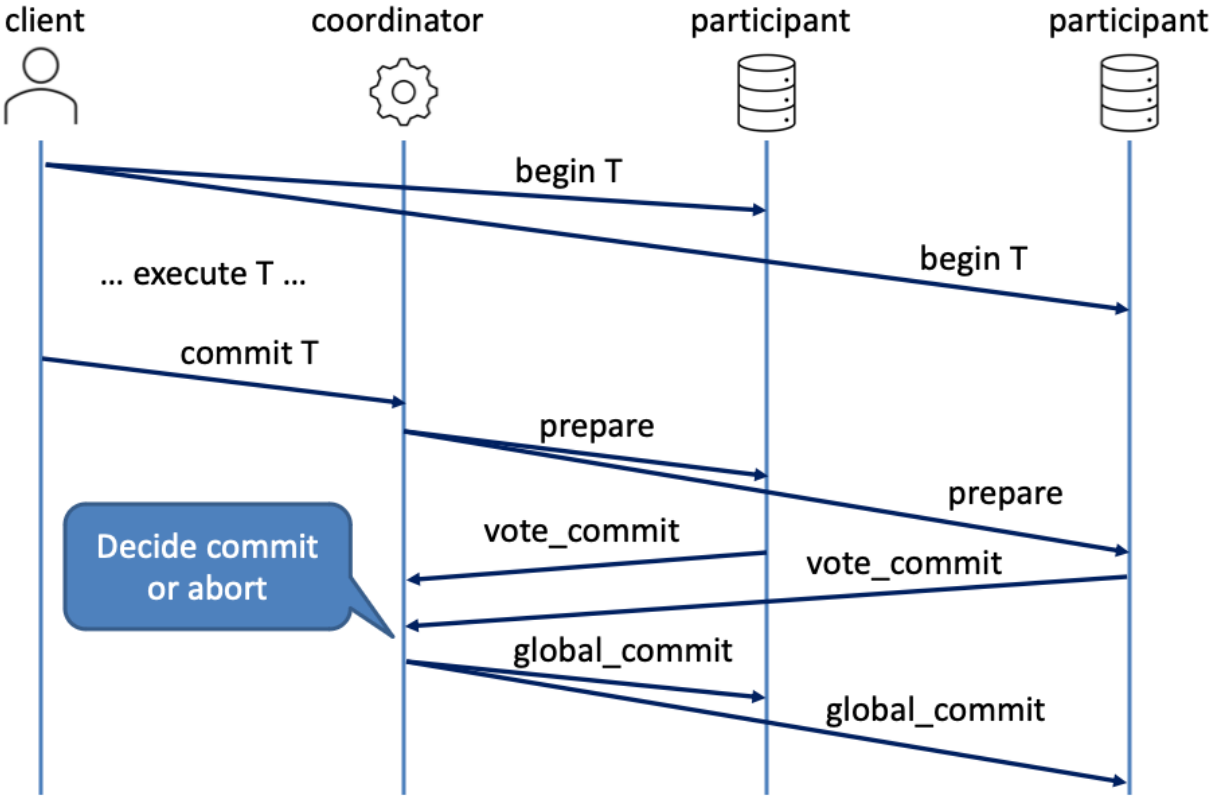
Consensus protocols	Atomic commit protocols
One or more nodes propose a value	Every node votes to commit or abort
Nodes agree on one of the proposed value	Commit if and only if all nodes vote to commit, otherwise abort
Tolerates failures, as long as a majority of nodes is available	Any crash leads to an abort

Two Phase Commit (2PC)

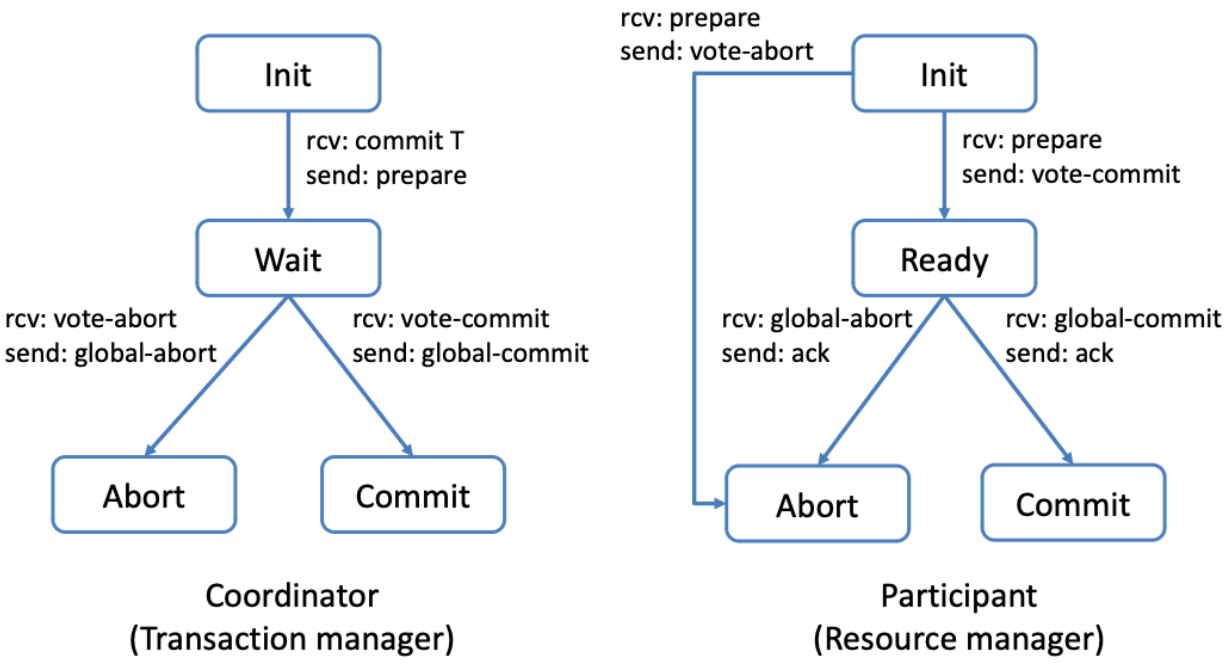
The Two Phase Commit protocol is a type of Atomic commitment protocol that consists of two main phases:

1. The **commit-request** phase (or **voting** phase): a coordinator process attempts to prepare all the transaction's participating processes (known as participants, cohorts or workers) to take the necessary steps for either committing or aborting the transaction and to vote either "Yes", as in commit the transaction, or "No", as in abort it

2. The **commit** phase: based on the voting of the participants, the coordinator decides whether to commit or to abort the transaction. A transaction is committed if and only if all workers voted "Yes". After the commit, the coordinator notifies all workers of the results of the operation. Then the participants follow with the needed actions with their local transactional resources.



Here is a breakdown of the workflows of both coordinators and participants in 2PC:



2PC is not 100% fault-tolerant however, and may fail under the following conditions:

- A participant fails to vote: after a timeout, the coordinator counts the vote of crashed workers as an abort
- The coordinator fails: there are two scenarios in this case:
  1. The coordinator fails while participants are preparing to vote: the transaction can still be aborted, since no participant has already received a global commit decision
  2. The coordinator fails while participants are waiting for a global decision: workers cannot decide to abort on their own in this case, so they must wait for the coordinator to get back online or ask another worker to take a decision for them, since another worker may have already received a reply from the coordinator or could also still be in its initial phase, so an abort is assumed.

If every worker is in the same `ready` state, nothing can be decided until the coordinator recovers, which means the whole system is stuck. We say that 2PC is a **blocking protocol**: it is vulnerable to single-node failure and if it takes some time to recover the failed coordinator, the whole system may remain unavailable.

Three Phase Commit (3PC)

The Three Phase Commit protocol attempts to solve the problems introduced by 2PC by adding a new phase to it, so that:

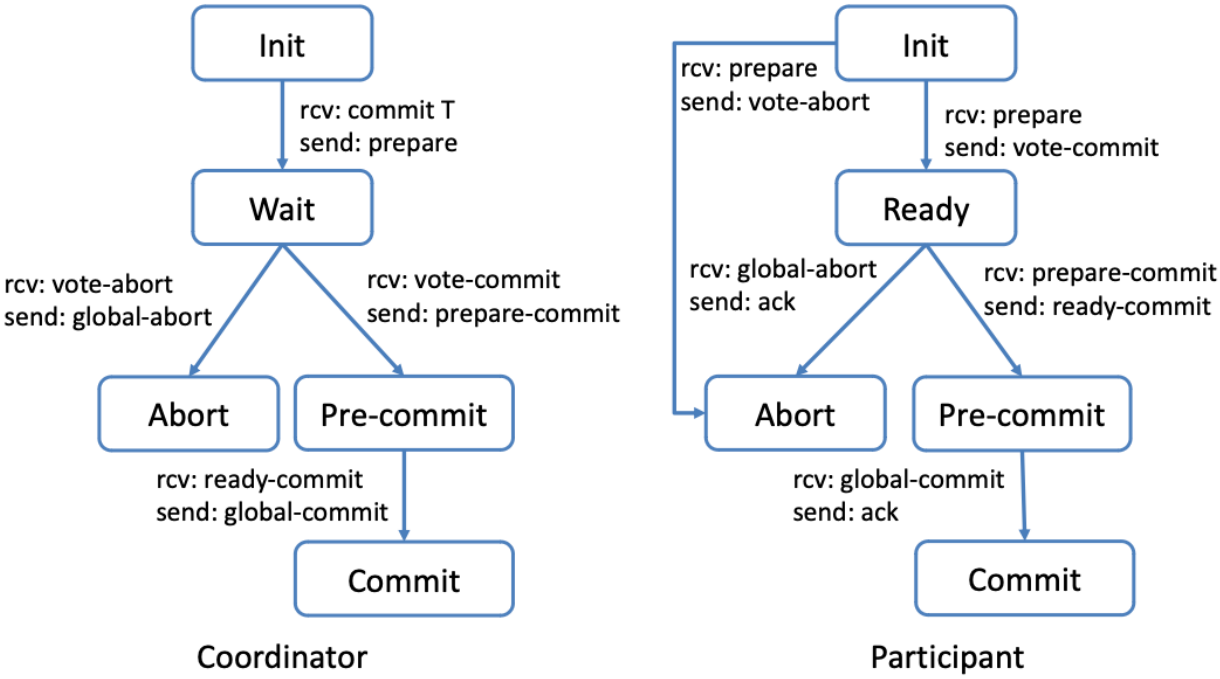
- There are no states leading directly to commit and abort
- There are no states where a final decision is impossible that lead to a commit

The above conditions ensure that 3PC is **non-blocking**, differently from 2PC.

The main idea behind 3PC is to add a new "Prepared to commit" state:

- If the coordinator fails before sending `preCommit` messages, the workers will unanimously agree that the operation was aborted
- The coordinator will not send out a `doCommit` message until all cohort members have acked that they are "Prepared to commit"

This eliminates the possibility that any cohort member actually completed the transaction before all cohort members were aware of the decision to do so.



Let's now see what happens in 3PC if:

- A participant fails:
  - If the coordinator is blocked waiting for the votes of the participants, it can assume an abort
  - If the coordinator is blocked in the pre-commit state, it can safely commit and tell the failed participant to commit when it recovers
- The coordinator fails:
  - If a participant is blocked waiting for the "Prepared to commit" state, it can decide to abort
  - If a participant is blocked waiting for a global decision, it can contact another participant to take a decision. According to what messages it receives, the participant can:
    - Abort, if at least one other participant does so
    - Commit, if at least one other participant does so
    - Abort, if another participant is stil in the initial phase
    - Commit, if the number of participants in the "Pre-commit" state and the number of participants in a "Ready" state form a majority
    - Abort, if the number of participants in a "Ready" state form a majority, but none of them are in "Pre-commit"

> **Note**
>
> In 3PC, two participants cannot be simultaneously in "Pre-commit" and "Init".

We can see that 3PC guarantees safety: it **never leads to an incorrect state**. If it is implemented in a synchronous system, it also guarantees liveness: it **never blocks if a majority of nodes are alive** and can communicate with each other.

In an asynchronous system, however, the protocol may not terminate.

Finally, 3PC is much more expensive than 2PC, since it requires three phases for each transaction instead of two.

The CAP theorem

Any distributed system where nodes share some replicated data can have **at most** two of these three desirable properties:

- **Consistency** (C) equivalent to having a single, up-to-date copy of the data
- High **availability** (A) of the data for updates (also known as *liveness*)
- Tolerance to network **partitions** (P)

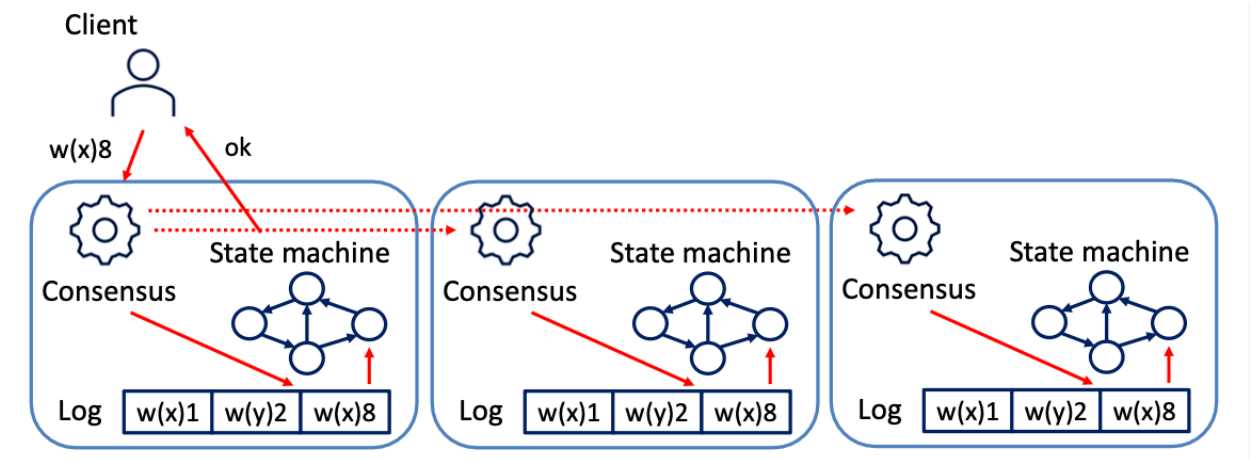
In the presence of network partitions, for example, one cannot have both perfect availability and consistency.

Modern data systems provide suitable balance of availability and consistency for the application at hand, e.g. using weaker definitions of consistency.

Replicated state machines

Replicated state machines are systems that have the same data replicated on every node that composes them but behave as a single machine from the user's perspective.

Thanks to a general purpose consensus algorithm, the nodes in a system work as a coherent group and operate on identical copies of the same state. Furthermore, the system offers continuous service, even if some of the nodes fail.



In order to give the user the illusion that they are communicating with a single system, a coordinator, or leader, is selected among the nodes. The user interactions happen with the leader first, and are then propagated to the other machines.

A replicated log also ensures that state machines execute the same commands in the same order.

We assume that replicated state machines systems use unreliable, asynchronous communication and that processes can fail. Processes must remember what they were doing before crashing, so their state is periodically recorded to durable storage.

With replicated state machines, we want to guarantee safety and liveness:

- All non-failing machines execute the same commands in the same order
- The system makes progress if any majority of machines are up and can communicate with one another

The latter point cannot be guaranteed in theory, according to the FLP theorem, but can be guaranteed in practice, under typical operating conditions.

Paxos protocol

Paxos has been the reference algorithm for consensus for about thirty years. It was proposed in 1989 and published in 1998.

Paxos had some problems however:

- Paxos could only bring machines to agree on a single decision, not on a sequence of requests (although multi-Paxos solved this problem)

- Paxos is very difficult to understand
- Paxos is difficult to use in practice, since there are no reference implementations and there is no agreement on the details of the protocol

Raft protocol

The Raft protocol is equivalent to multi-Paxos in terms of assumptions, guarantees and performance. Its design goal is **understandability**: it is easy to explain, understand, use and adapt. Raft has several reference implementations.

Raft can be decomposed in three main aspects:

- **Log replication**
  - The leader accepts commands from clients and appends them to its log
  - The leader then replicates its log to other servers
- **Leader election**
  - One server is selected to act as a leader
  - When the current leader crashes, a new election is started
- **Safety**
  - The logs must be kept consistent
  - Only servers with up-to-date logs can become leaders

In Raft, all nodes can be in one of the following three states:

- Follower
- Leader
- Candidate

All nodes start as followers.

If followers do not hear from a leader for a while, they become candidates. A candidate can run an election: if it wins, it then becomes the leader.

Normal operation

Normally, Raft operates like this:

1. A client sends a command to the leader
2. The leader appends that command to its log
3. The leader sends an `AppendEntries` to all followers

4. Once the new entry is committed, the leader executes the command in its state machine and returns the result to the client
5. The leader then notifies followers of the committed entries in the subsequent `AppendEntries` messages
6. Followers execute the committed commands in their state machines

If some followers crash or are slowed down, the leader retries to send its `AppendEntries` message until it succeeds. The optimal performance is achieved when one sent message is successful for any majority of followers.

Terms and elections

The leader periodically sends an `AppendEntries` message with unacked entries to all followers. If, however, followers do not hear from the leader for a predefined amount of time, the leader is assumed dead and they start an election.

The timeout for the election is usually in a range of 150 to 300 ms and is randomised, in order not to have too many parallel elections.

Raft divides time into **terms** of arbitrary lengths when performing an election. All terms are numbered with consecutive integers and each server maintains a *current term* value, exchanged in every communication during the election.

Terms are needed to identify obsolete information: if a message is sent with an old current term value, it is discarded.

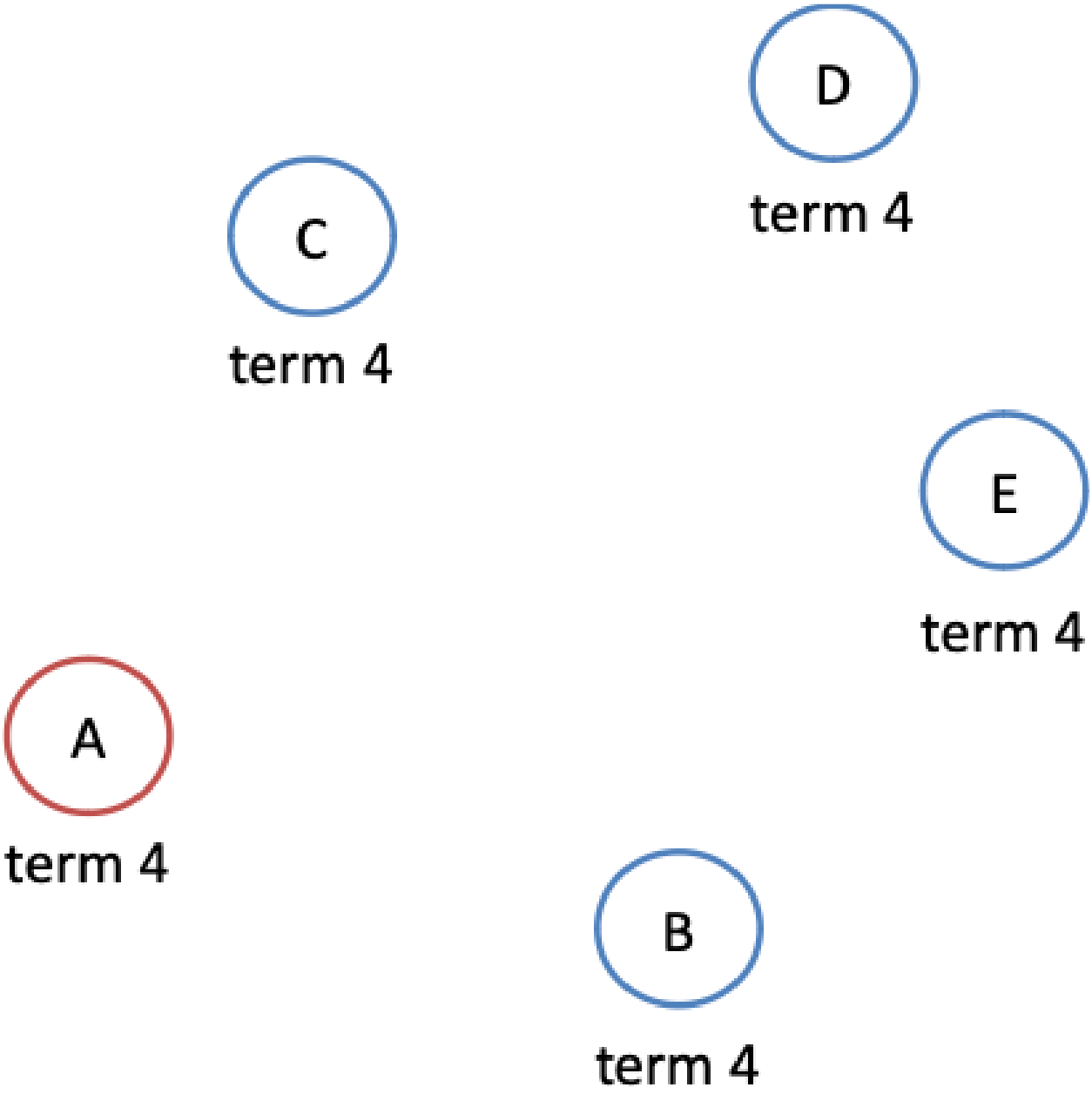
Each term begins with an election, in which one or more candidates try to become the leader. However, there is at most one leader per term. If there is no majority for any candidate in a given election, followers will hold another election once the next timeout expires.

The fact that only one winner is allowed per term guarantees safety: since only the leader receives commands from clients, having only one leader ensures that commands are executed in the same order on every node.

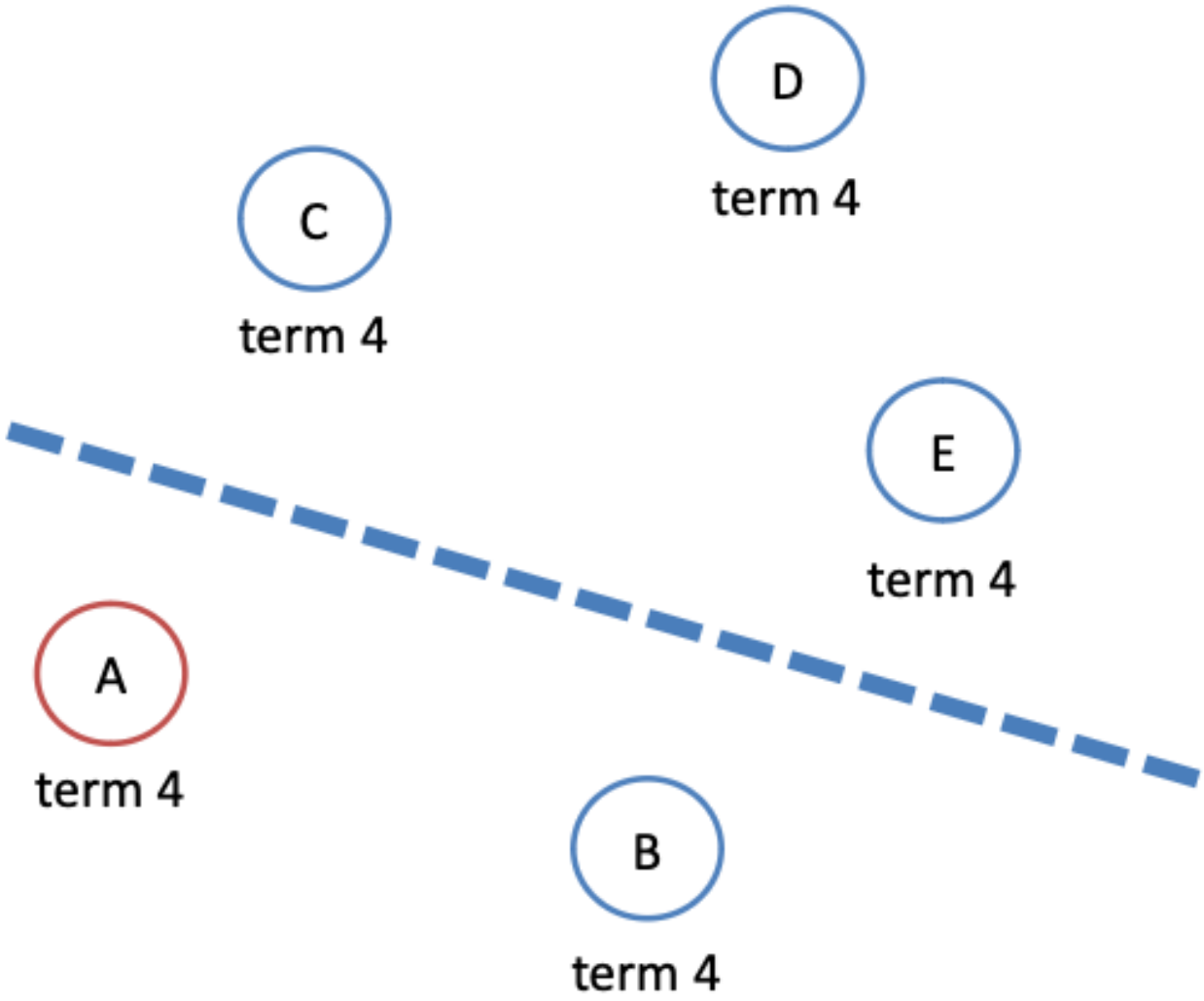
Furthermore, the fact that one candidate must eventually win helps the system's liveness. The election system works very well if the timeout for the elections is much larger than the average communication RTT between nodes.

Raft and network partitions

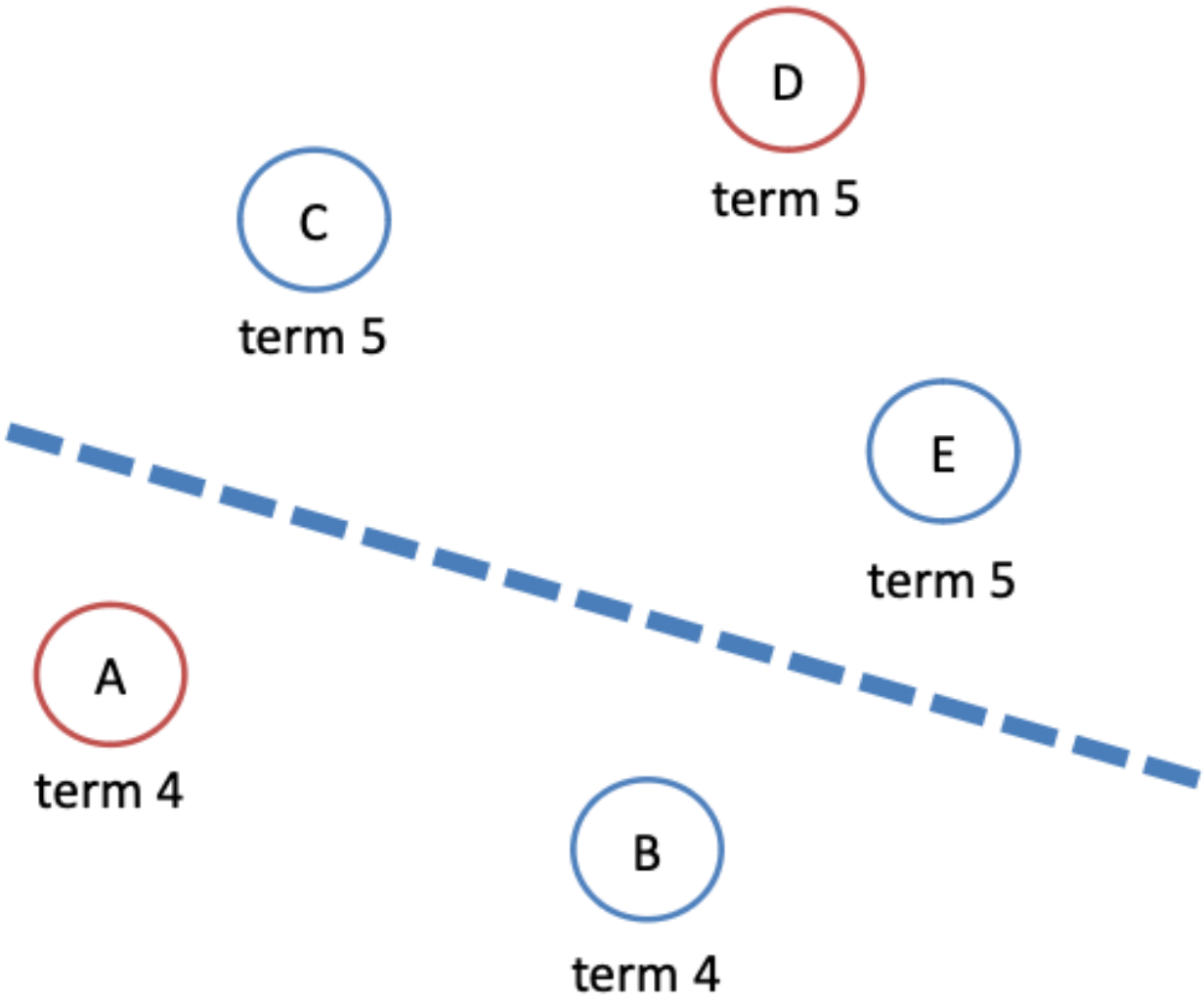
Raft works well with network partitions too. Let's take the following 5-node network as an example:



If we introduce a network partition that divides nodes A and B from C, D and E:



The leader A will eventually become unreachable for C, D and E and they will start a new election. Let's suppose they elect D as their new leader:

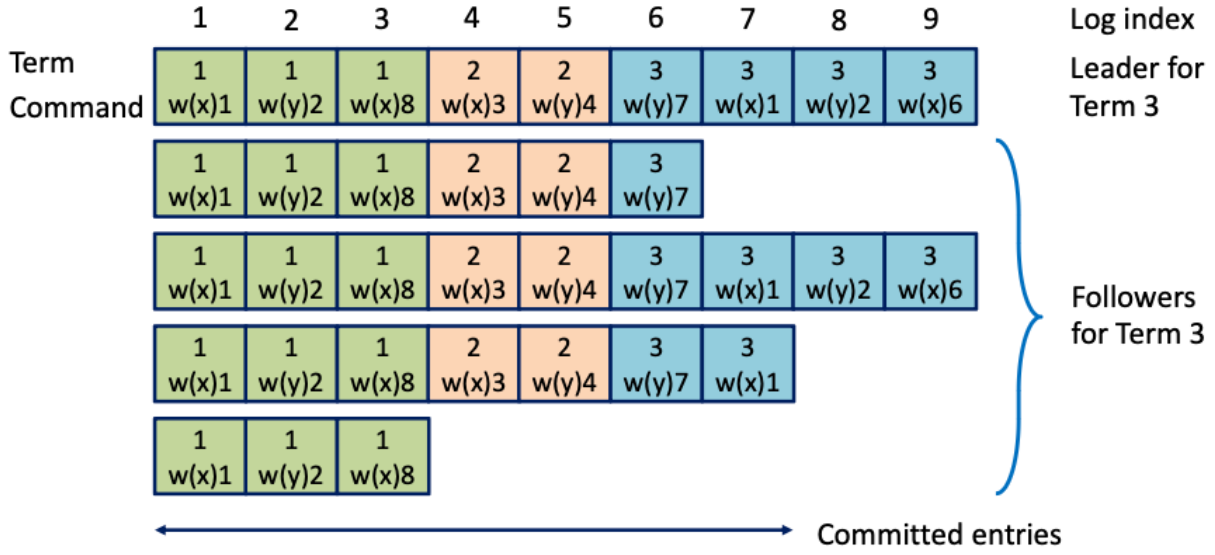


Clients may still now send commands to A, but since it cannot reach a majority, it won't be able to commit them. However, if a command is sent to D, it can reach a majority and commit the command, so no time is lost.

Once the network partition is removed, A will eventually receive a message with the current term value of 5 and understand that he is not up to date anymore. He will then step back as the leader and let D manage this term.

Raft log structure

Let's now take a look at a normal log structure in Raft:



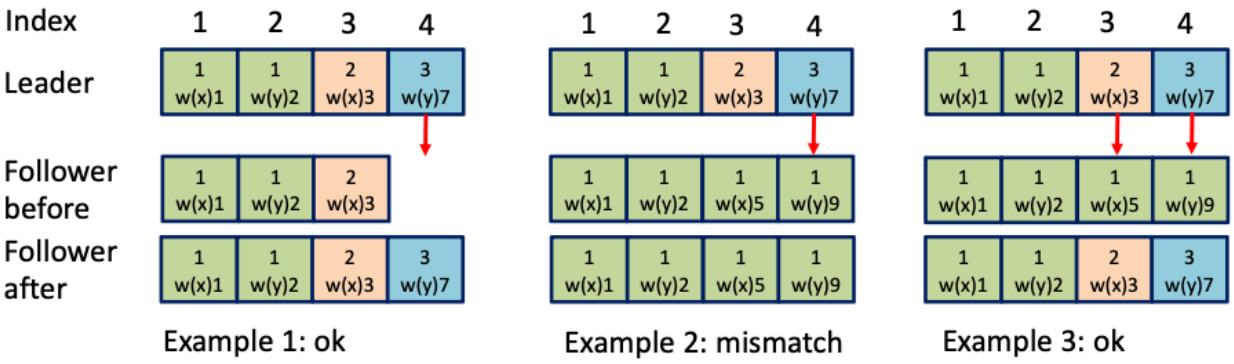
Logs are always stored on disk, so that they survive process failures.

Since node crashes can result in log inconsistencies, Raft just assumes that the leader's log is always correct and normal operation fixes inconsistencies by replicating the leader's log to all nodes.

Raft guarantees the **log matching property**, which means that the following statements are always true:

1. If log entries on different servers have the same index and term, then they store the same command and the logs are identical in all preceding entries
2. If a given entry is committed, all preceding entries are also committed

Raft performs consistency checks by sending the index and term values of the entry preceding the new one with every `AppendEntries` message. Every follower that receives the message must contain matching entries, otherwise the message is rejected and the leader retries with lower log indexes.



In example two we can see that the message is rejected because the operation with index 4 does not have the correct preceding operation with index 3 in the follower

log. In example three this mismatch is fixed by substituting all mismatching entries, that is indexes 3 and 4, since indexes 1 and 2 were correct.

Safety: leader completeness

Once a log entry is committed, all future leaders must store that entry, in order to ensure safety. This also means that servers with incomplete logs **must not get elected**.

Candidates always include, with every `RequestVote` message, index and term of their last log entry. The voting server denies the vote if its log is more up to date than the one contained in the message. In this way, no out-of-date leader can be elected.

Communication with clients

In Raft, clients always interact with the leader. When a client starts, it connects to a random server, which in turn communicates to the client the current leader.

If the leader crashes, the communication with the client times out.

Raft guarantees linearisable semantics: all operations behave as if they were executed once and only once on a single copy.

In case of a leader failure, a client may need to retry to send a message. To avoid duplicates, the client attaches a sequential identifier of the request to every message and the server stores the last identifier for each client, as part of their log.

Use in distributed DBMSs

Some modern distributed DBMSs integrate replicated state machines and commit protocols.

Generally, in big systems, the database is partitioned and each partition is replicated. This means that inter-partition transactions must guarantee atomic commitment and replicated state machines must guarantee that all replicas process the same sequence of operations.

We can solve both these problems by applying both commit protocols and replicated state machine protocols:

- Each partition is managed by a set of machines that behave as one, via a replicated state machine
- 2PC executes atomic commit across partitions

Byzantine conditions

State machine replication can be extended to consider byzantine processes. This is known as **Byzantine Fault Tolerant replication** (BFT replication).

BFT replication starts from the same assumptions as Paxos and Raft, but requires $3f + 1$ participants to tolerate f failing nodes.

We will not consider BFT replication protocols in this course, but we'll take a look at a similar and strictly related technology.

Blockchains

Cryptocurrencies can be seen as replicated state machines in which the state is the current balance of each user.

The state of each machine is stored in a replicated ledger, the node's log, which records all operations.

This is a byzantine environment: there may be malicious users that may try to "double spend" their money, creating inconsistent copies of the log.

In blockchains, we have the following assumptions:

- Very large number of nodes
- The set of participating nodes is unknown upfront
- No single entity owns the majority of compute resources

In blockchains there are **no provable safety guarantees**, although they are very likely to hold.

We will now refer to permissionless systems based on proof of work, the likes of Bitcoin. Different approaches may differ in terms of both assumptions and guarantees.

Bitcoin blockchain

In the Bitcoin system, the blockchain is the public ledger that records transactions. It contains all transactions from the beginning of the blockchain and each block of the chain includes multiple transactions. It is essentially a distributed ledger, a replicated log of all transactions.

All transactions in the blockchain are also digitally signed. If the user's private key is lost, bitcoins are lost as well.

Adding a block to the chain requires solving a mathematical problem: this is the proof of work mechanism. The input of the problem to solve is usually the existing chain and the new block that needs to be added. This links the block to the chain.

The solutions to this type of problems are very hard to find, which makes it very difficult to brute force them. Problem difficulty is also adjusted in order to adapt to current computational power. On average, right now, one solution to these problems is found every 10 minutes.

Solutions, however, are very easy to verify, which enables a quick validity check.

Proof of work is computed by special nodes, called **miners**, which collect new transactions into a block. When a miner finds the proof, it broadcasts the new block, which defines the next block of valid transactions. The other miners receive that block and try to create the next one, and so on.

It is very unlikely that two miners find the same solution to a problem at the same time. If it were to happen though, the solution that includes more blocks survives.

Someone with enough computational power and one bitcoin could, theoretically, create two concurrent chains however. In this way, a user could spend his money twice, and no user can ever be 100% sure that another, conflicting chain does not exist.